hw9_JavonTimmerberg

April 10, 2024

1 Programming Section

1.1 Creating Dataset

Before constructing the recurrent neural network, we first had to process the dataset we were provided into embeddings that would allow our network to function without relying on large one hot vectors to account for the vocabulary size of whatever dataset is being trained and tested on. These embeddings were generated through the DistilBert model, a lightweight transformer. These embeddings were created first at the single word level and then afterwards the DistilBert library was used to create n-gram tokens and their respective embeddings.

After generating the word embeddings, the respective sentiment classes for each review were converted into one hot vectors.

The embedding tensors and the sentiment onehot vectors were stored in a list and then saved to the disk to be used for the dataloader.

1.1.1 Generating Word Embeddings

[ ]:
```
import pandas as pd
from transformers import DistilBertTokenizer, DistilBertModel

data = pd.read_csv('/content/drive/MyDrive/ECE60146/HW9/data.csv')
# Split columns into lists
column_lists = [data[col].tolist() for col in data.columns]
sentences = column_lists[0]
sentiments = column_lists[1]

"""
Tokenization procedure from HW9 pdf Code
"""

word_tokenized_sentences = [sentence.split() for sentence in sentences]

max_len = max([len(sentence) for sentence in word_tokenized_sentences])
padded_sentences = [sentence + ['[PAD]'] * (max_len - len(sentence)) for
                    sentence in word_tokenized_sentences]

vocab = {}
```

```
vocab['[PAD]'] = 0
for sentence in padded_sentences:
  for token in sentence:
    if token not in vocab:
      vocab[token] = len(vocab)

padded_sentences_ids = [[vocab[token] for token in sentence] for
                        sentence in padded_sentences]

word_embeddings = []
for tokens in padded_sentences_ids:
  input_ids = torch.tensor(tokens).unsqueeze(0)
  with torch.no_grad():
    outputs = distilbert_model(input_ids)
  word_embeddings.append(outputs.last_hidden_state)

tensors_cpu = [tensor.cpu() for tensor in word_embeddings]

# Save the list of tensors to a file
torch.save(tensors_cpu, '/content/drive/MyDrive/ECE60146/HW9/
 ↪word_embeddings_tensor.pt')
```

1.1.2 Generating Subword Embeddings

[ ]:
```
"""
Tokenization procedure from HW9 pdf Code
"""

model_ckpt = "distilbert-base-uncased"
distilbert_tokenizer = DistilBertTokenizer.from_pretrained(model_ckpt)

bert_tokenized_sentences_ids = [distilbert_tokenizer.encode(sentence,
                                    padding='max_length', truncation=True,
                                    max_length=max_len) for sentence in␣
 ↪sentences]

bert_tokenized_sentences_tokens = [distilbert_tokenizer.
 ↪convert_ids_to_tokens(sentence)
                                     for sentence in␣
 ↪bert_tokenized_sentences_ids]

model_name = 'distilbert/distilbert-base-uncased'
distilbert_model = DistilBertModel.from_pretrained(model_name)

subword_embeddings = []
for tokens in bert_tokenized_sentences_ids:
  input_ids = torch.tensor(tokens).unsqueeze(0)
```

```
  with torch.no_grad():
    outputs = distilbert_model(input_ids)
  subword_embeddings.append(outputs.last_hidden_state)

tensors_cpu = [tensor.cpu() for tensor in subword_embeddings]

# Save the list of tensors to a file
torch.save(tensors_cpu, '/content/drive/MyDrive/ECE60146/HW9/
 ↪subword_embeddings_tensor.pt')
```

1.1.3 Generating Sentiment One Hot Vectors

[ ]:
```
encoded_sents = [0 if sentiment == 'negative' \
                 else 1 if sentiment == 'neutral' \
                 else 2 for sentiment in sentiments]

# Using pytorch function to generate one hot vectors from encoded sentiment list
onehot_sents = [F.one_hot(torch.tensor(encoded_sent), num_classes=3) for␣
 ↪encoded_sent in encoded_sents]
tensors_cpu = [tensor.cpu() for tensor in onehot_sents]

# Save the list of tensors to a file
torch.save(tensors_cpu, '/content/drive/MyDrive/ECE60146/HW9/sentiments.pt')
```

1.2 Dataset Class

After generating the review and sentiment tensors from data.csv, the lists were loaded into a dataset class which then split 80% of the dataset to be used in training and 20% of the dataset to be used in testing.

[ ]:
```
class SentimentDataset(torch.utils.data.Dataset):
  def __init__(self, train=True, device=torch.device("cuda:0" if torch.cuda.
 ↪is_available() else "cpu"),
               embeddings_file='/content/drive/MyDrive/ECE60146/HW9/
 ↪subword_embeddings_tensor.pt',
               sentiments_file='/content/drive/MyDrive/ECE60146/HW9/sentiments.
 ↪pt'):

    # Load previously created embeddings and sentiment tensors
    embeddings = torch.load(embeddings_file)
    sentiments = torch.load(sentiments_file)
    self.train = train
    self.device = device
    self.tot_length = len(sentiments)
    # Assuming sentiments are uniformly distributed
    # through the dataset, split dataset 80:20 for training:testing
    train_length = int(self.tot_length // 1.25)
```

```
    if train == True:
      self.length = train_length
      self.embeddings = embeddings[:train_length]
      self.sentiments = sentiments[:train_length]
    else:
      self.length = len(sentiments[train_length:])
      self.embeddings = embeddings[train_length:]
      self.sentiments = sentiments[train_length:]

  def __len__(self):
    return self.length

  def __getitem__(self, index):
    # return embedding and sentiment tensors
    return torch.squeeze(self.embeddings[index].to(self.device)), self.
↪sentiments[index].to(self.device)
```

1.3 Network

To predict the sentiment of the review, the following network was used. The main points of the network are the GRU layer which processes the embedding sequence for a linear layer which predicts the sentiment of the text. The LogSoftmax allows the output to be interpreted as the log probabilities for each class of sentiment. This is then combined with the NLLLoss criterion used in the training procedure.

[ ]:
```
class GRUnet(nn.Module):
  def __init__(self, input_size, hidden_size, output_size, n_layers,␣
↪drop_prob=0.2, bidirectional=False,
               epochs=5, lr=1e-4, device=torch.device("cuda:0" if torch.cuda.
↪is_available() else "cpu")):
    """
    Network design adjusted from to include bidirectional setup:
    https://engineering.purdue.edu/kak/distDLS/DLStudio-2.4.3.html
    """
    super().__init__()
    self.bidirectional = bidirectional
    self.hidden_size = hidden_size
    self.n_layers = n_layers
    self.gru = nn.GRU(input_size, hidden_size, n_layers, batch_first=True,␣
↪dropout=drop_prob, bidirectional=bidirectional)
    linear_hidden = hidden_size
    if self.bidirectional:
      linear_hidden = linear_hidden * 2
    self.fc = nn.Linear(linear_hidden, output_size)
    self.relu = nn.ReLU()
    self.logsoftmax = nn.LogSoftmax(dim=1)
    self.epochs = epochs
```

```
    self.lr = lr
    self.device = device

  def forward(self, x, h):
    out, h = self.gru(x, h)
    out = self.fc(self.relu(out[:,-1]))
    out = self.logsoftmax(out)
    return out, h

  def init_hidden(self, batch_size):
    weight = next(self.parameters()).data
    layers = self.n_layers
    if self.bidirectional:
      layers = layers * 2
    hidden = weight.new(layers, batch_size, self.hidden_size).zero_()
    return hidden
```

1.4 Sentiment Analysis Results

The training procedure shown below was adjusted from the DLStudio library. The main adjustment to the code was to use weights with the loss criterion. During my initial training rounds, my network would just learn to only predict neutral. When watching the outputs from the network and the associated classes from each input, I noticed the training dataset featured a much larger amount of neutral sentiment reviews. After multiple epochs, the network would converge to a local minima where the inputted review would have little effect on what the output log probabilities were. The learning rate, number of layers and the size of hidden input to the linear layer had no effect on this convergence. The large number of neutral sentiment reviews far outnumbered the amount of positive and especially the amount of negative reviews in the dataset. Any adjustments to the parameters made by classes that weren't neutral would be far outweighed by the number of neutral sentiments which would push the network to the local minima. To account for this, I added weights to the criterion which would account for the imbalance in the dataset so that each occurrrence of a negative review would have the same overall weight as four occurrences of a neutral review, and two occurrences for a positive review.

The training loss results for the subword and word embeddings of both the unidirectional and bidirectional networks are shown below in figures 1-4. All the trained network losses leveled out within 5 epochs of the training datasets but the bidirectional networks took longer to reach their minimums.

1.4.1 Training Procedure Code

[ ]:
```
def run_code_for_training_for_text_classification_with_GRU(net,␣
↪train_dataloader, net_num):
    """
    Training code adjusted from:
    https://engineering.purdue.edu/kak/distDLS/DLStudio-2.4.3.html
    """
```

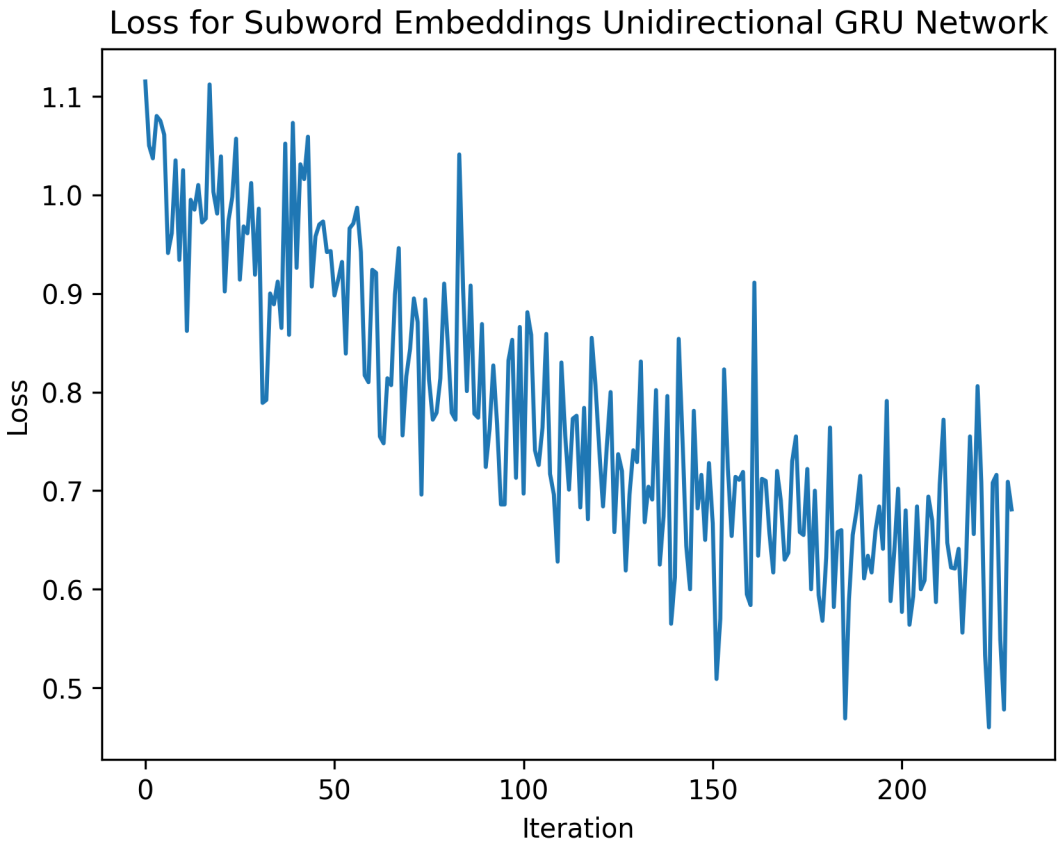
```
    filename_for_out = "/content/drive/MyDrive/ECE60146/HW9/
↪performance_numbers_" + str(net.epochs) + str(net_num) +".txt"
    FILE = open(filename_for_out, 'w')
    net = net.to(net.device)
    # Added weights to NLLLoss algorithm to account for training set imbalance.
    # Weights are determined by ratios between sentiments
    # Negative: 675
    # Neutral: 2516
    # Positive: 1482
    criterion = nn.NLLLoss(weight = torch.tensor([1, 0.25, .5]).to(net.device))
    optimizer = optim.Adam(net.parameters(), lr=net.lr)
    negative_total = 0
    neutral_total = 0
    positive_total = 0
    for epoch in range(net.epochs):
      print("")
      running_loss = 0.0
      start_time = time.time()
      for i, data in enumerate(train_dataloader):
        review_tensor,sentiment = data
        review_tensor = review_tensor.to(net.device)
        sentiment = sentiment.to(net.device)
        optimizer.zero_grad()
        hidden = net.init_hidden(1).to(net.device) ## (A)
        output, hidden = net(review_tensor, hidden) ## (C)
        gt_idx = torch.argmax(sentiment).item()
        # Recording counts of sentiments for network weights
        if gt_idx == 0:
          negative_total += 1
        elif gt_idx == 1:
          neutral_total += 1
        elif gt_idx == 2:
          positive_total += 1
        # print(output, sentiment)
        loss = criterion(output, torch.unsqueeze(torch.argmax(sentiment), 0))
        running_loss += loss.item()
        loss.backward()
        optimizer.step()
        if i % 100 == 99:
          avg_loss = running_loss / float(99)
          current_time = time.time()
          time_elapsed = current_time-start_time
          print("[epoch:%d iter:%3d elapsed_time: %d secs] loss: %.3f" %␣
↪(epoch+1,i+1, time_elapsed,avg_loss))
          FILE.write("%.3f\n" % avg_loss)
          FILE.flush()
          running_loss = 0.0
```

```
        # print(negative_total, neutral_total, positive_total)
    torch.save(net.state_dict(), "/content/drive/MyDrive/ECE60146/HW9/net" +␣
 ↪str(net_num))
```

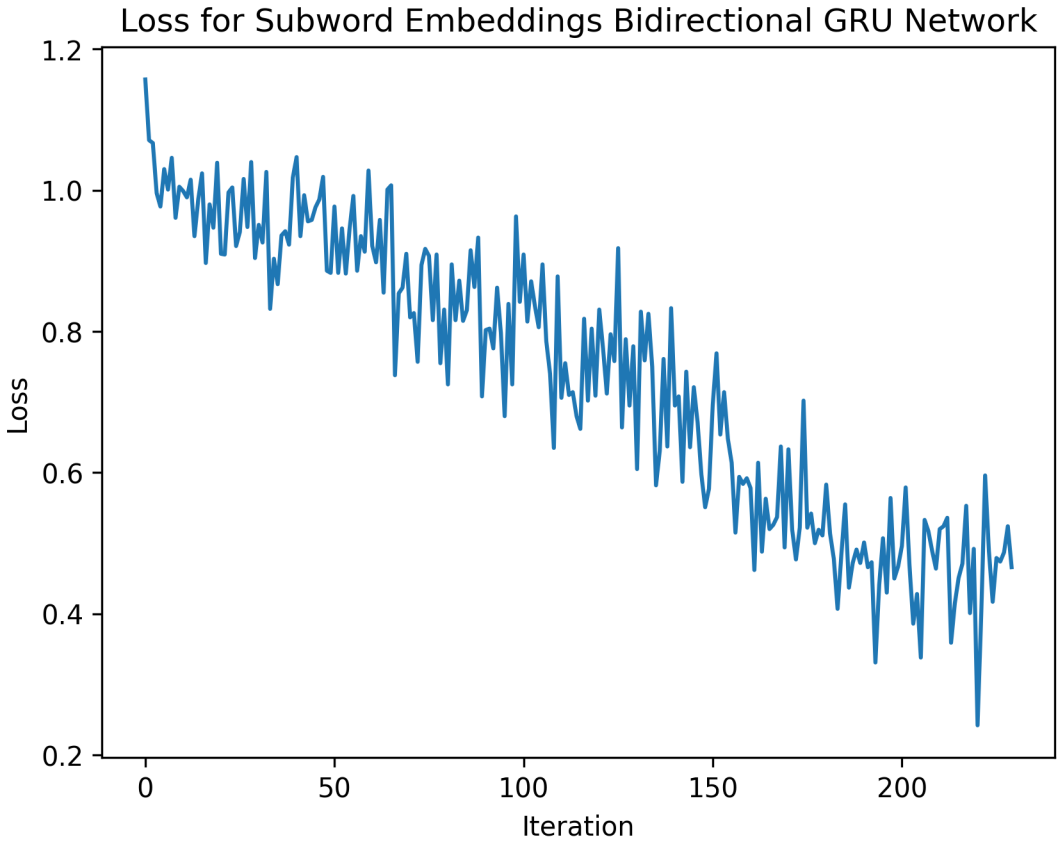
[ ]:
```
# Create dataloaders for subword embeddings
train_set = SentimentDataset()
test_set = SentimentDataset(train=False)

train_loader = DataLoader(train_set, batch_size=1, num_workers=0, shuffle=True)
test_loader = DataLoader(test_set, batch_size=1, num_workers=0, shuffle=True)

hidden_size=2048
n_layers=2

# Train subword sentiment networks
net = GRUnet(input_size=768, hidden_size=hidden_size, output_size=3,␣
 ↪n_layers=n_layers, bidirectional=False)
run_code_for_training_for_text_classification_with_GRU(net, train_loader, 0)
run_code_for_testing_text_classification_with_GRU(net, test_loader, 0)

net = GRUnet(input_size=768, hidden_size=hidden_size, output_size=3,␣
 ↪n_layers=n_layers, bidirectional=True)
run_code_for_training_for_text_classification_with_GRU(net, train_loader, 1)
run_code_for_testing_text_classification_with_GRU(net, test_loader, 1)

# Create dataloaders for whole word embeddings
word_train_set = SentimentDataset(embeddings_file='/content/drive/MyDrive/
 ↪ECE60146/HW9/word_embeddings_tensor.pt')
word_test_set = SentimentDataset(embeddings_file='/content/drive/MyDrive/
 ↪ECE60146/HW9/word_embeddings_tensor.pt',train=False)
word_train_loader = DataLoader(train_set, batch_size=1, num_workers=0,␣
 ↪shuffle=True)
word_test_loader = DataLoader(test_set, batch_size=1, num_workers=0,␣
 ↪shuffle=True)

# Train whole word setiment networks
net = GRUnet(input_size=768, hidden_size=hidden_size, output_size=3,␣
 ↪n_layers=n_layers, bidirectional=False)
run_code_for_training_for_text_classification_with_GRU(net, word_train_loader,␣
 ↪2)
run_code_for_testing_text_classification_with_GRU(net, word_test_loader, 2)

net = GRUnet(input_size=768, hidden_size=hidden_size, output_size=3,␣
 ↪n_layers=n_layers, bidirectional=True)
run_code_for_training_for_text_classification_with_GRU(net, word_train_loader,␣
 ↪3)
```

```
run_code_for_testing_text_classification_with_GRU(net, word_test_loader, 3)
```

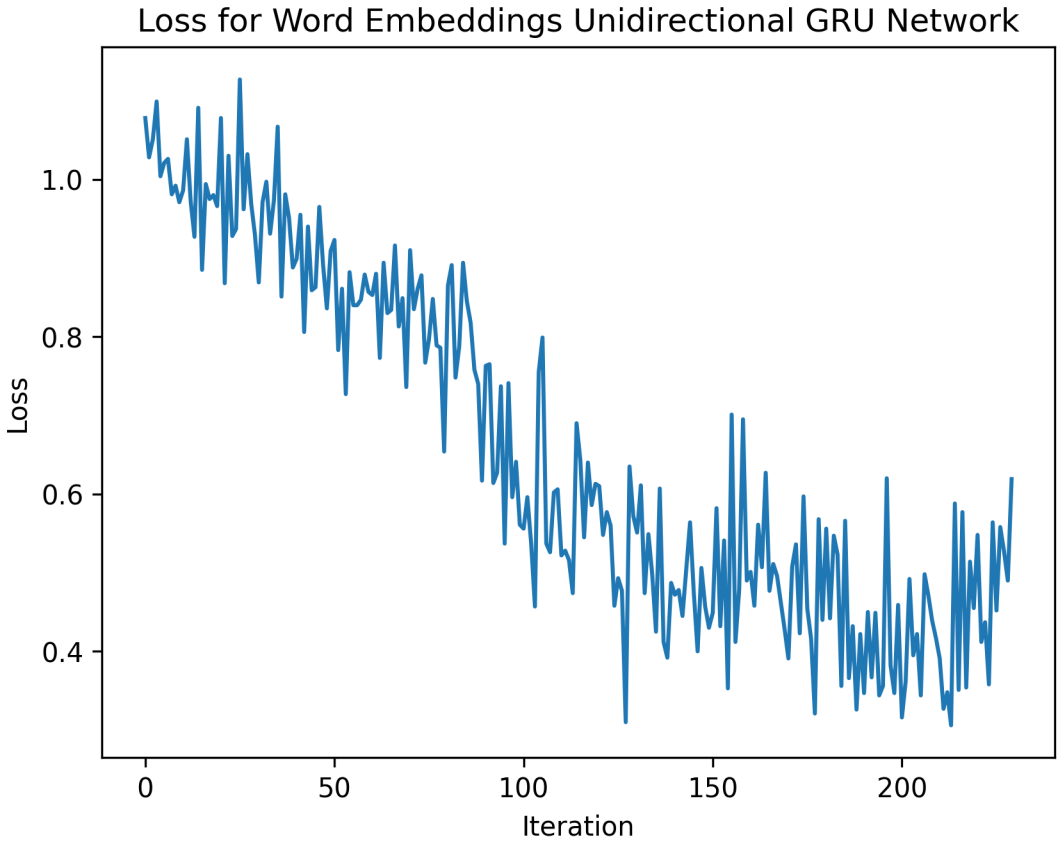
1.4.2 Figure 1. Training Loss Results from Unidirectional Network from Subwords Encodings generated from data.csv



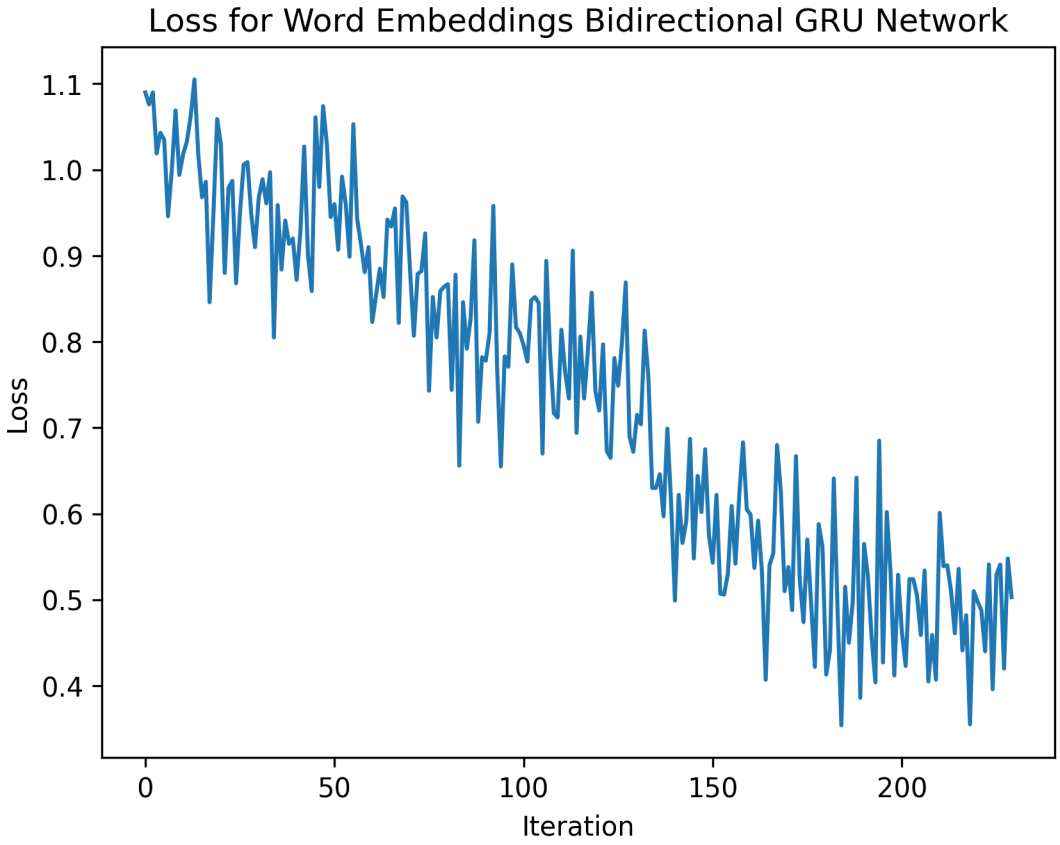
1.4.3 Figure 2. Training Loss Results from Bidirectional Network on Subwords Encodings generated from data.csv



1.4.4 Figure 3. Training Loss Results from Unidirectional Network on Whole Word Encodings generated from data.csv



1.4.5 Figure 4. Training Loss Results from Bidirectional Network on Whole Word Encodings generated from data.csv



1.4.6 Testing Procedure Code

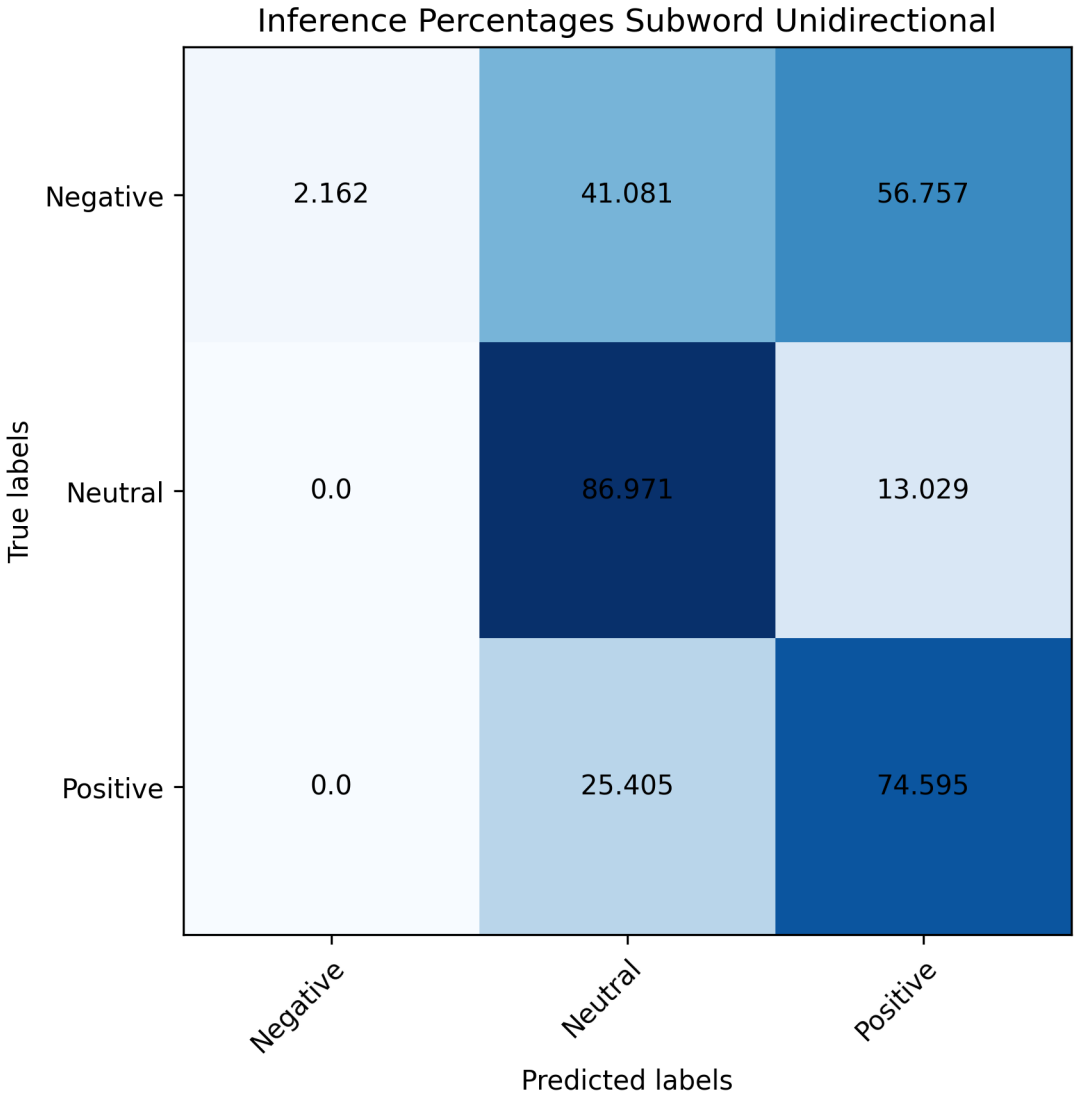
After training each of the networks, the following testing procedure was run to see the accuracy over the testing dataset. The best performing network was the Unidirectional network trained on the full word embeddings with an average sentiment accuracy of 75%. The worst performing network was the Unidirectional network trained on the subword embeddings with an average sentiment accuracy of 54%. While the total accuracy of the network would be higher, this would be skewed by the ability of the network to prioritize predicting a neutral sentiment from the largest class of the testing dataset. On average the Bidirectional networks performed better than the Unidirectional networks but the average accuracy difference was only 2%.

[ ]:
```
def run_code_for_testing_text_classification_with_GRU(net, test_dataloader,␣
↪net_num):
    """
    Testing code adjusted from: https://engineering.purdue.edu/kak/distDLS/
↪DLStudio-2.4.3.html
    """
    net.load_state_dict(torch.load("/content/drive/MyDrive/ECE60146/HW9/net" +␣
↪str(net_num)))
```

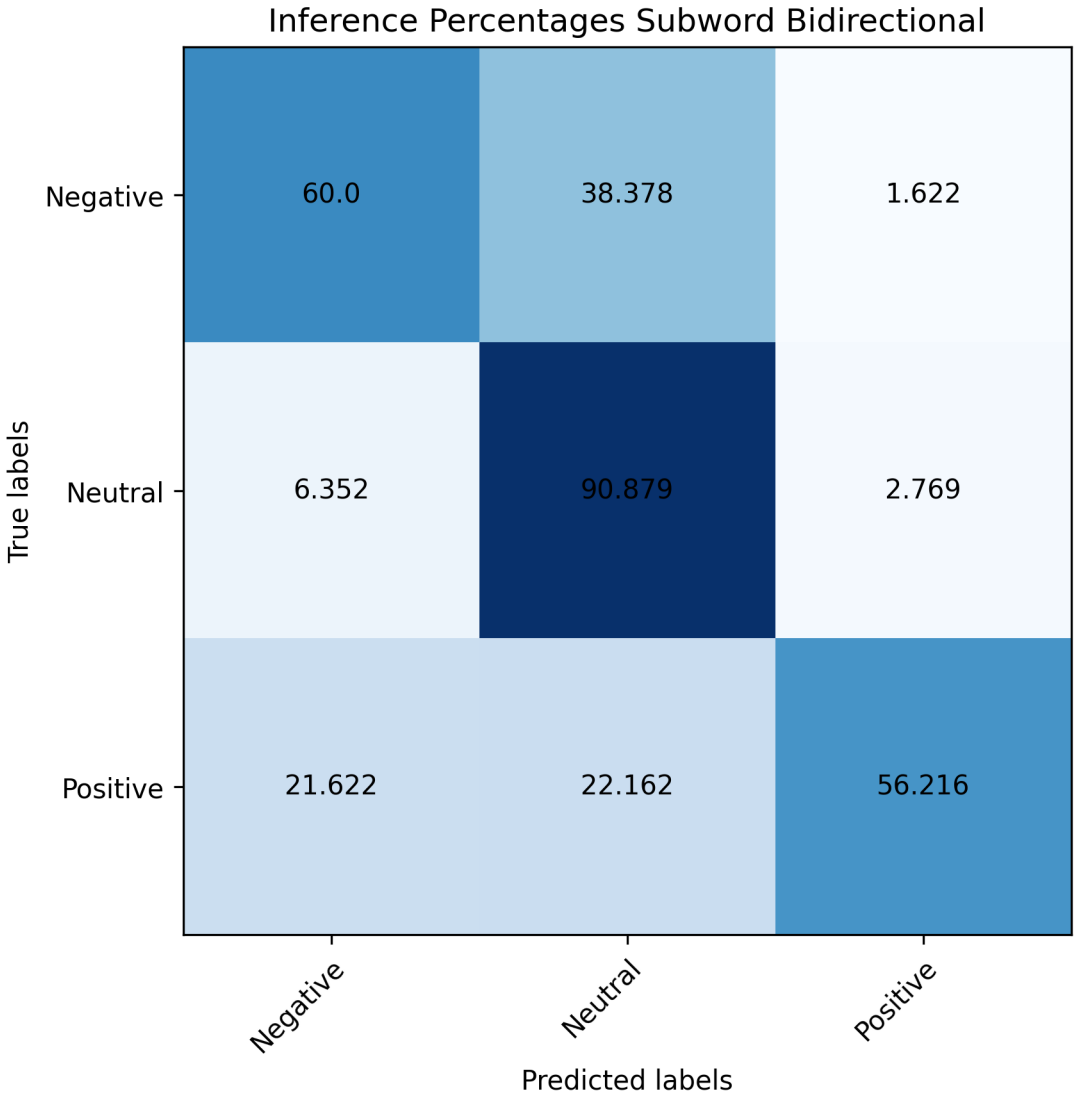
```
net = net.to(net.device)
classification_accuracy = 0.0
negative_total = 0
neutral_total = 0
positive_total = 0
confusion_matrix = torch.zeros(3,3)
with torch.no_grad():
  for i, data in enumerate(test_dataloader):
    review_tensor,sentiment = data
    hidden = net.init_hidden(1).to(net.device) ## (A)

    output, hidden = net(review_tensor, hidden) ## (C)
    # print(review_tensor.shape)
    predicted_idx = torch.argmax(output).item()
    gt_idx = torch.argmax(sentiment).item()
    if i % 100 == 99:
      print(" [i=%d] predicted_label=%d gt_label=%d\n\n" % (i+1,␣
↪predicted_idx,gt_idx))
    if predicted_idx == gt_idx:
      classification_accuracy += 1
    if gt_idx == 0:
      negative_total += 1
    elif gt_idx == 1:
      neutral_total += 1
    elif gt_idx == 2:
      positive_total += 1
    confusion_matrix[gt_idx,predicted_idx] += 1

# Create confusion matrix of inference percentages
out_percent = np.zeros((3,3), dtype='float')
out_percent[0,0] = "%.3f" % (100 * confusion_matrix[0,0] /␣
↪float(negative_total))
out_percent[0,1] = "%.3f" % (100 * confusion_matrix[0,1] /␣
↪float(negative_total))
out_percent[0,2] = "%.3f" % (100 * confusion_matrix[0,2] /␣
↪float(negative_total))
out_percent[1,0] = "%.3f" % (100 * confusion_matrix[1,0] /␣
↪float(neutral_total))
out_percent[1,1] = "%.3f" % (100 * confusion_matrix[1,1] /␣
↪float(neutral_total))
out_percent[1,2] = "%.3f" % (100 * confusion_matrix[1,2] /␣
↪float(neutral_total))
out_percent[2,0] = "%.3f" % (100 * confusion_matrix[2,0] /␣
↪float(positive_total))
out_percent[2,1] = "%.3f" % (100 * confusion_matrix[2,1] /␣
↪float(positive_total))
```

```
 out_percent[2,2] = "%.3f" % (100 * confusion_matrix[2,2] /␣
↪float(positive_total))

 print("\n\nNumber of positive reviews tested: %d" % positive_total)
 print("\n\nNumber of neutral reviews tested: %d" % neutral_total)
 print("\n\nNumber of negative reviews tested: %d" % negative_total)
 print("\n\nDisplaying the confusion matrix:\n")
 out_str = " "
 out_str += "%18s %18s %18s" % ('predicted negative', 'predicted␣
↪neutral','predicted positive')
 print(out_str + "\n")
 for i,label in enumerate(['true negative', 'true neutral ', 'true positive']):
   out_str = "%12s: " % label
   for j in range(3):
     out_str += "%18s" % out_percent[i,j]
   print(out_str)
```

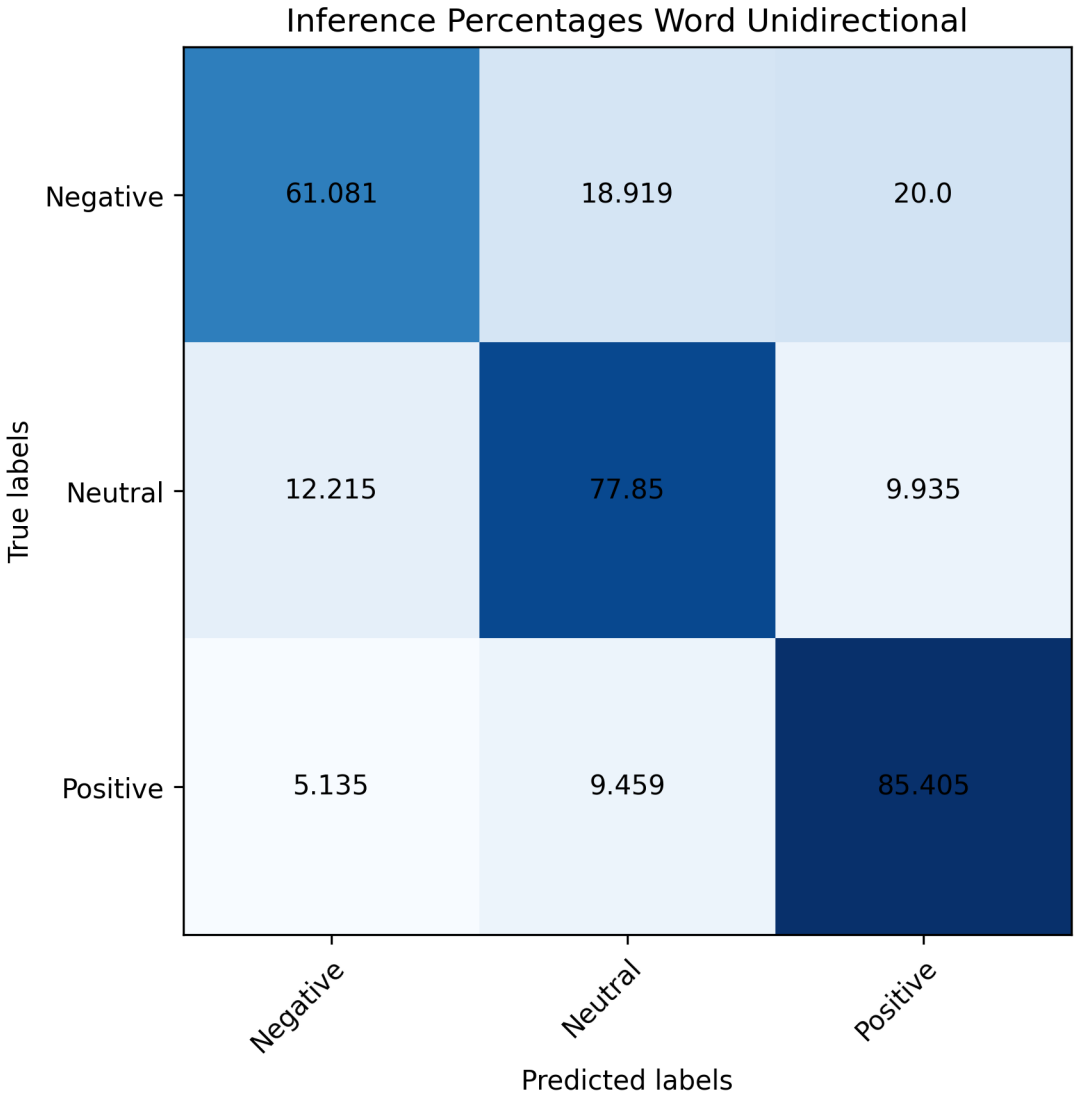
1.4.7 Figure 5. Testing Results Confusion Matrix for Unidirectional Subword Embeddings Network



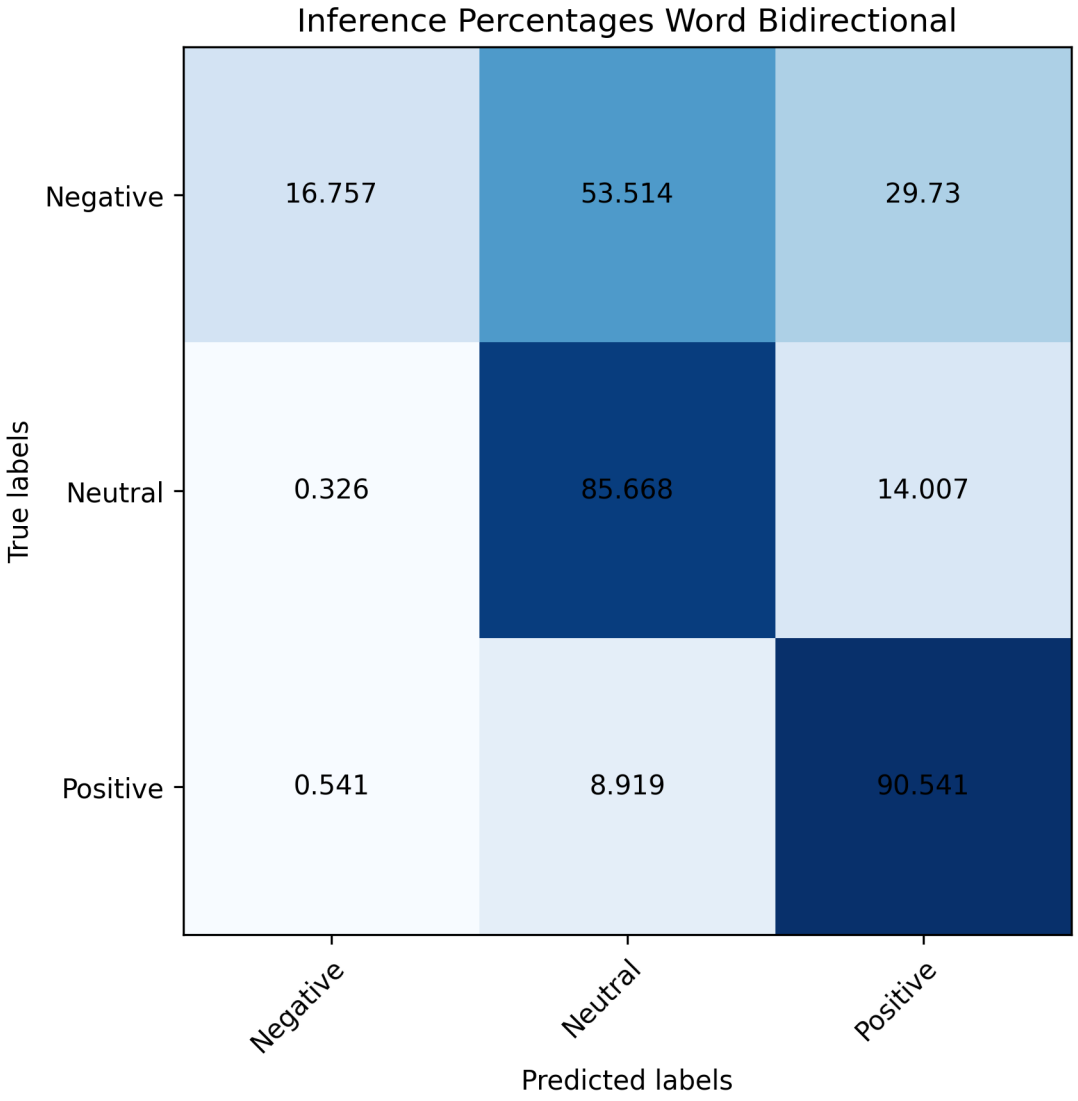
1.4.8 Figure 6. Testing Results Confusion Matrix for Bidirectional Subword Embeddings Network



1.4.9 Figure 7. Testing Results Confusion Matrix for Unidirectional Word Embeddings Network



1.4.10 Figure 8. Testing Results Confusion Matrix for Bidirectional Word Embeddings Network



2 Extra Credit

After performing the testing and training procedure using the data.csv dataset, the process was repeated using the text classification dataset from DLStudio. The overall design of the network stayed the same but the procedure was adjusted to use the dataloader from the DLStudio library. The network was adjusted to have a smaller hidden size to save on training time but the training loss was much more variable with respect to the average loss during training than the original network. The average accuracy for both the unidirectional and the bidirectional networks were 80.5% but the best performing network was the bidirectional network. The worst performing still had an average accuracy of 79%. Throughout both experiments, while the bidirectional networks outperformed the unidirectional networks, the improvement was minimal.

2.0.1 Training Procedure Code

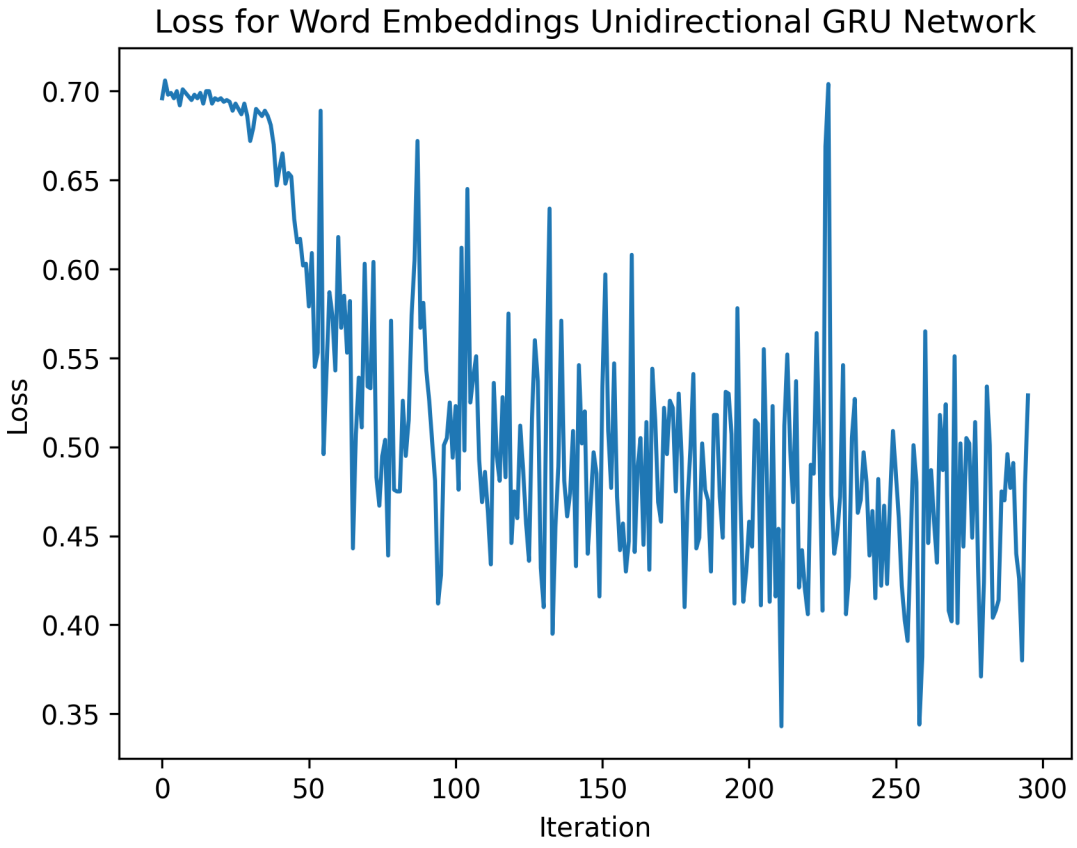
```
[ ]: def run_code_for_training_for_text_classification_with_GRU_extra(net,␣
 ↪train_dataloader, net_num):
        """
        Code adjusted from: https://engineering.purdue.edu/kak/distDLS/DLStudio-2.4.
 ↪3.html
        """
        filename_for_out = "/content/drive/MyDrive/ECE60146/HW9/
 ↪performance_numbers_" + str(net.epochs) + str(net_num) +".txt"
        FILE = open(filename_for_out, 'w')
        net = net.to(net.device)
        criterion = nn.NLLLoss()
        optimizer = optim.Adam(net.parameters(), lr=net.lr)
        negative_total = 0
        neutral_total = 0
        positive_total = 0
        for epoch in range(net.epochs):
          print("")
          running_loss = 0.0
          start_time = time.time()
          for i, data in enumerate(train_dataloader):
            review_tensor,_,sentiment = data['review'], data['category'],␣
 ↪data['sentiment']
            review_tensor = review_tensor.to(net.device)
            sentiment = sentiment.to(net.device)
            optimizer.zero_grad()
            hidden = net.init_hidden(1).to(net.device) ## (A)
            output, hidden = net(review_tensor, hidden) ## (C)
            gt_idx = torch.argmax(sentiment).item()
            if gt_idx == 0:
              negative_total += 1
            elif gt_idx == 1:
              neutral_total += 1
            elif gt_idx == 2:
              positive_total += 1
            # print(output, sentiment)
            loss = criterion(output, torch.unsqueeze(torch.argmax(sentiment), 0))
            running_loss += loss.item()
            loss.backward()
            optimizer.step()
            if i % 100 == 99:
              avg_loss = running_loss / float(99)
              current_time = time.time()
              time_elapsed = current_time-start_time
              print("[epoch:%d iter:%3d elapsed_time: %d secs] loss: %.3f" %␣
 ↪(epoch+1,i+1, time_elapsed,avg_loss))
```

```
            FILE.write("%.3f\n" % avg_loss)
            FILE.flush()
            running_loss = 0.0
        # print(negative_total, neutral_total, positive_total)
    torch.save(net.state_dict(), "/content/drive/MyDrive/ECE60146/HW9/net" +␣
 ↪str(net_num))
```

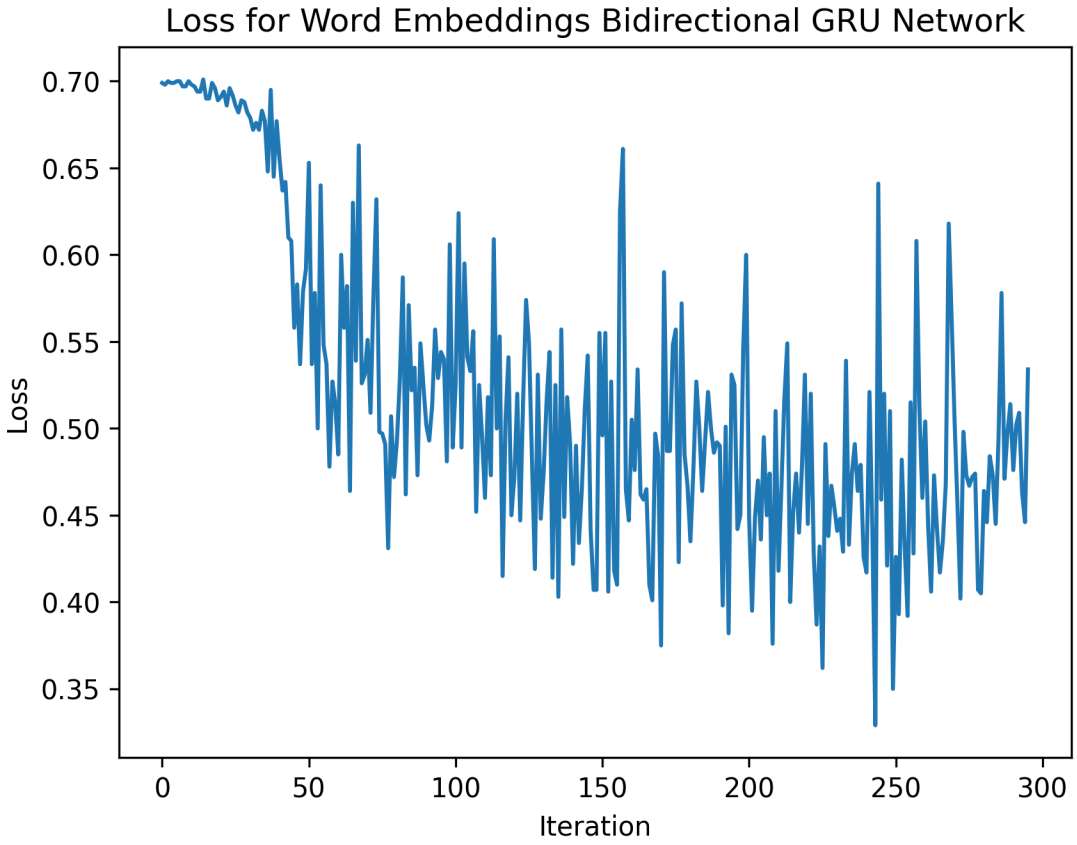
[ ]:
```
"""
Code adjusted from: https://engineering.purdue.edu/kak/distDLS/DLStudio-2.4.3.
 ↪html
"""
from DLStudio import *
dataroot = "/content/drive/MyDrive/ECE60146/HW9/data/"


dataset_archive_train = "sentiment_dataset_train_200.tar.gz"
#dataset_archive_train = "sentiment_dataset_train_200.tar.gz"

dataset_archive_test =  "sentiment_dataset_test_200.tar.gz"
#dataset_archive_test = "sentiment_dataset_test_200.tar.gz"

path_to_saved_embeddings = "/content/drive/MyDrive/ECE60146/HW9/word2vec/"
#path_to_saved_embeddings = "./data/TextDatasets/word2vec/"


dls = DLStudio(
                  dataroot = dataroot,
                  path_saved_model = "/content/drive/MyDrive/ECE60146/HW9/data/
 ↪dlstudio_model_1",
                  momentum = 0.9,
                  learning_rate =  1e-5,
                  epochs = 1,
                  batch_size = 1,
                  classes = ('negative','positive'),
                  use_gpu = True,
              )

dataserver_train = DLStudio.TextClassificationWithEmbeddings.
 ↪SentimentAnalysisDataset(
                                train_or_test = 'train',
                                dl_studio = dls,
                                dataset_file = dataset_archive_train,
                                path_to_saved_embeddings =␣
 ↪path_to_saved_embeddings,
                    )
```

```
train_200_dataloader = torch.utils.data.DataLoader(dataserver_train,
                        batch_size=1,shuffle=True, num_workers=1)

hidden_size=400
epochs=4
net = GRUnet(input_size=300, hidden_size=hidden_size, epochs=epochs,␣
 ↪output_size=2, lr=1e-5, n_layers=n_layers, bidirectional=False)
run_code_for_training_for_text_classification_with_GRU_extra(net,␣
 ↪train_200_dataloader, 10)
net1 = GRUnet(input_size=300, hidden_size=hidden_size, lr=1e-5, epochs=epochs,␣
 ↪output_size=2, n_layers=n_layers, bidirectional=True)
run_code_for_training_for_text_classification_with_GRU_extra(net1,␣
 ↪train_200_dataloader, 11)
```

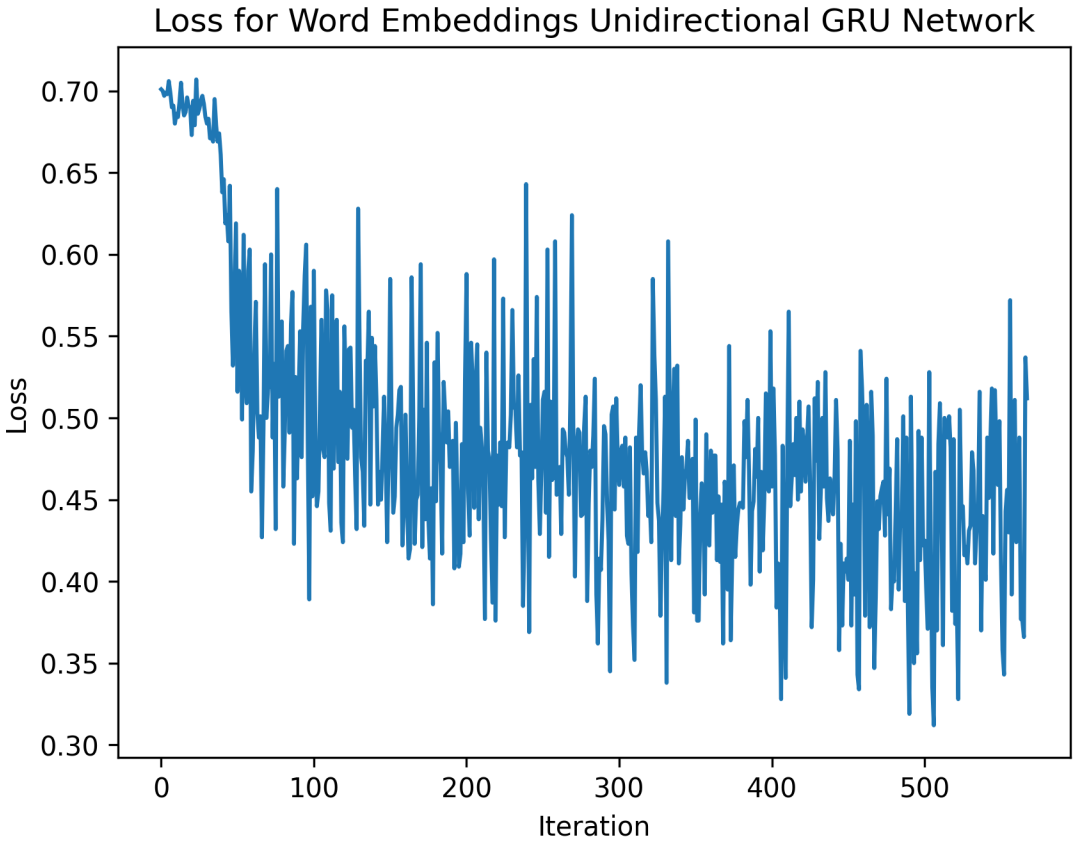
2.0.2 Figure 9. Training Loss Results from Unidirectional Network with 200 Embeddings



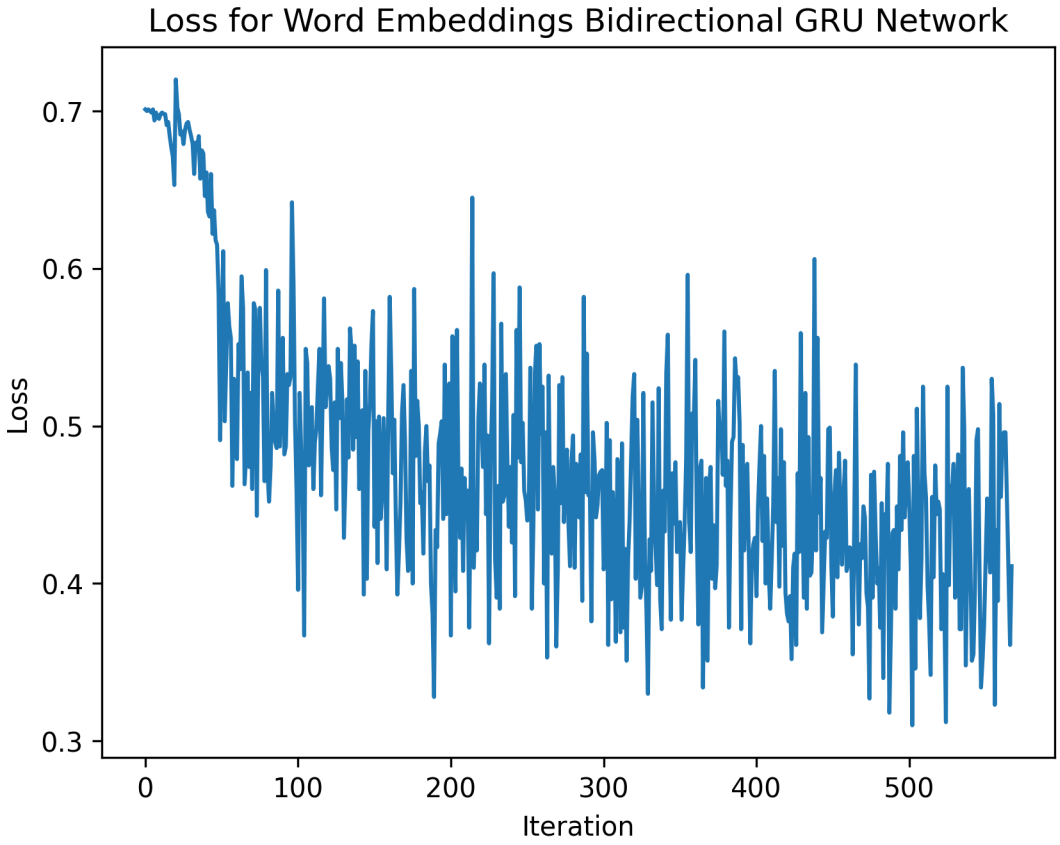
2.0.3 Figure 10. Training Loss Results from Bidirectional Network with 200 Embeddings



2.0.4 Figure 11. Training Loss Results from Unidirectional Network with 400 Embeddings



2.0.5 Figure 12. Training Loss Results from Unidirectional Network with 400 Embeddings



2.0.6 Testing Procedure Code

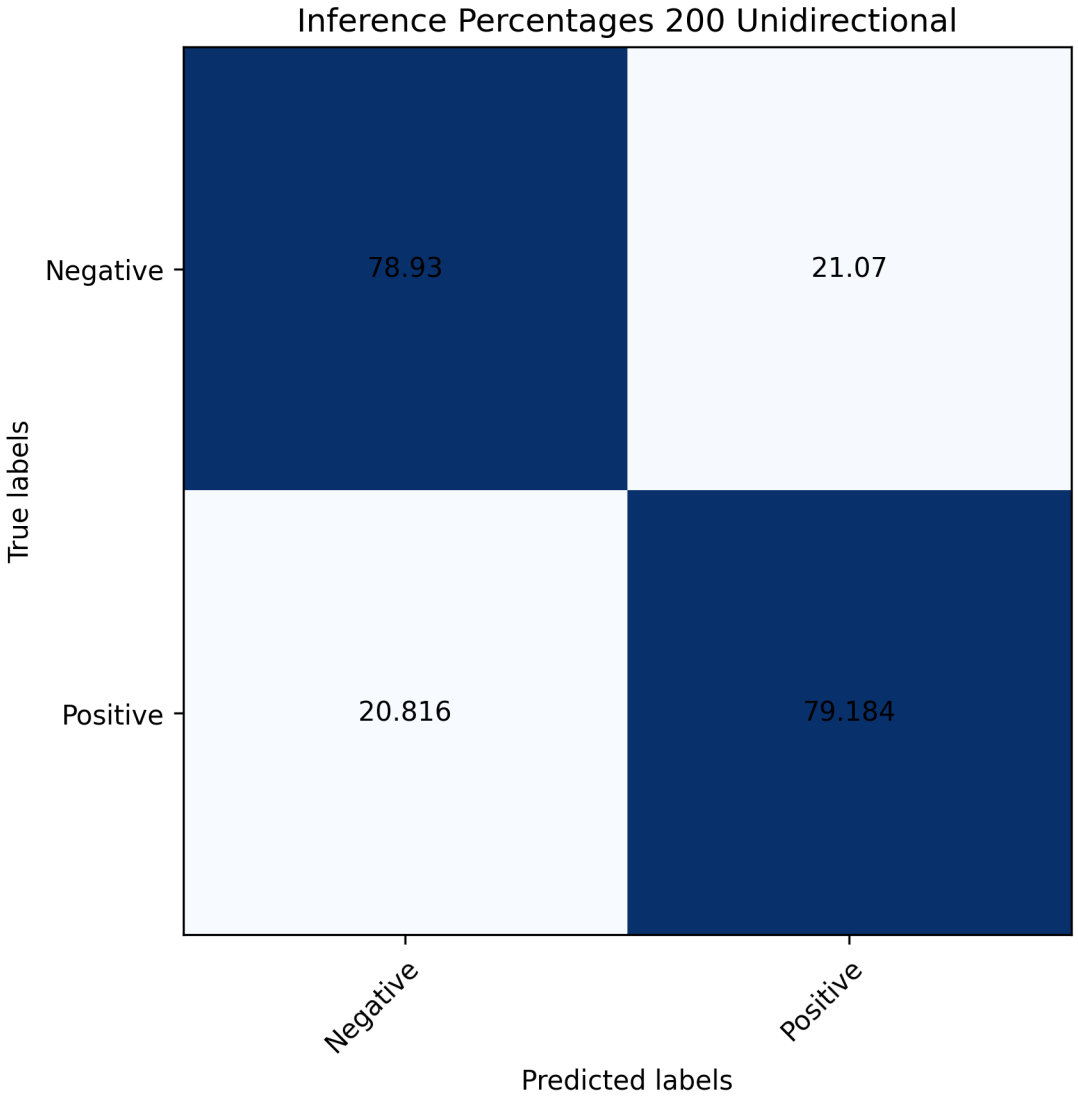
[ ]:
```
"""
Code adjusted from: https://engineering.purdue.edu/kak/distDLS/DLStudio-2.4.3.
 ↪html
"""

def run_code_for_testing_text_classification_with_GRU_extra(net,␣
 ↪test_dataloader, net_num):
  net.load_state_dict(torch.load("/content/drive/MyDrive/ECE60146/HW9/net" +␣
 ↪str(net_num)))
  classification_accuracy = 0.0
  negative_total = 0
  positive_total = 0
  confusion_matrix = torch.zeros(2,2)
  with torch.no_grad():
    for i, data in enumerate(test_dataloader):
      review_tensor,_,sentiment = data['review'], data['category'],␣
 ↪data['sentiment']
```

```
        hidden = net.init_hidden(1).to(net.device) ## (A)
        review_tensor = review_tensor.to(net.device)
        sentiment = sentiment.to(net.device)
        output, hidden = net(review_tensor, hidden) ## (C)
        # print(review_tensor.shape)
        predicted_idx = torch.argmax(output).item()
        gt_idx = torch.argmax(sentiment).item()
        if i % 100 == 99:
          print(" [i=%d] predicted_label=%d gt_label=%d\n\n" % (i+1,␣
 ↪predicted_idx,gt_idx))
        if predicted_idx == gt_idx:
          classification_accuracy += 1
        if gt_idx == 0:
          negative_total += 1
        else:
          positive_total += 1

        confusion_matrix[gt_idx,predicted_idx] += 1
    out_percent = np.zeros((3,3), dtype='float')
    out_percent[0,0] = "%.3f" % (100 * confusion_matrix[0,0] /␣
 ↪float(negative_total))
    out_percent[0,1] = "%.3f" % (100 * confusion_matrix[0,1] /␣
 ↪float(negative_total))
    out_percent[1,0] = "%.3f" % (100 * confusion_matrix[1,0] /␣
 ↪float(positive_total))
    out_percent[1,1] = "%.3f" % (100 * confusion_matrix[1,1] /␣
 ↪float(positive_total))


    print("\n\nNumber of positive reviews tested: %d" % positive_total)
    print("\n\nNumber of negative reviews tested: %d" % negative_total)
    print("\n\nDisplaying the confusion matrix:\n")
    out_str = " "
    out_str += "%18s %18s" % ('predicted negative','predicted positive')
    print(out_str + "\n")
    for i,label in enumerate(['true negative', 'true positive']):
      out_str = "%12s: " % label
      for j in range(2):
        out_str += "%18s" % out_percent[i,j]
      print(out_str)
```

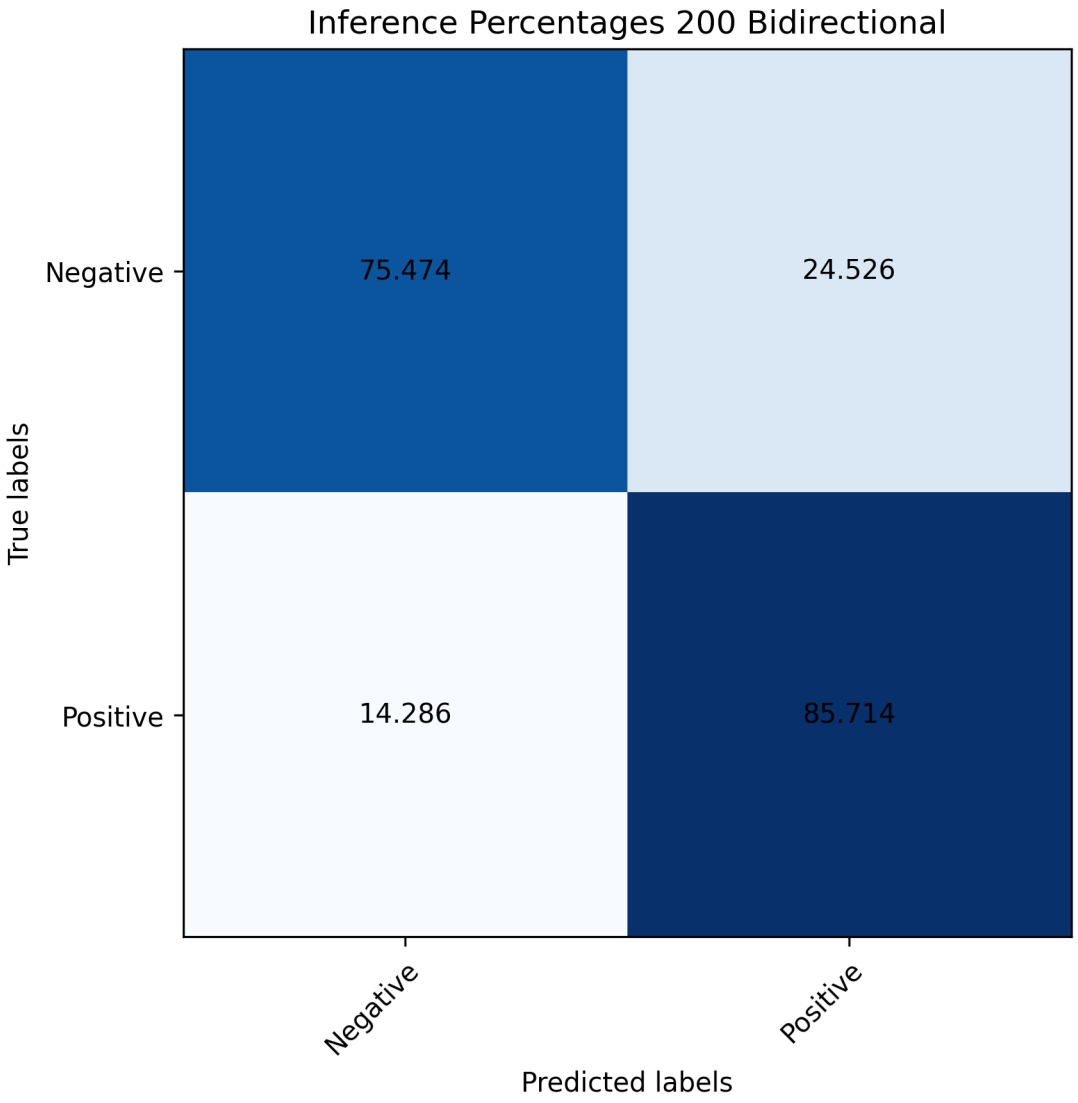
[ ]:
```
"""
Code adjusted from: https://engineering.purdue.edu/kak/distDLS/DLStudio-2.4.3.
 ↪html
"""
dataserver_test = DLStudio.TextClassificationWithEmbeddings.
 ↪SentimentAnalysisDataset(
```

```
                                train_or_test = 'test',
                                dl_studio = dls,
                                dataset_file = dataset_archive_test,
                                path_to_saved_embeddings =␣
 ↪path_to_saved_embeddings,
                )
test_200_dataloader = torch.utils.data.DataLoader(dataserver_test,
                    batch_size=1,shuffle=False, num_workers=1)
run_code_for_testing_text_classification_with_GRU_extra(net,␣
 ↪test_200_dataloader, 10)
run_code_for_testing_text_classification_with_GRU_extra(net1,␣
 ↪test_200_dataloader, 11)
```

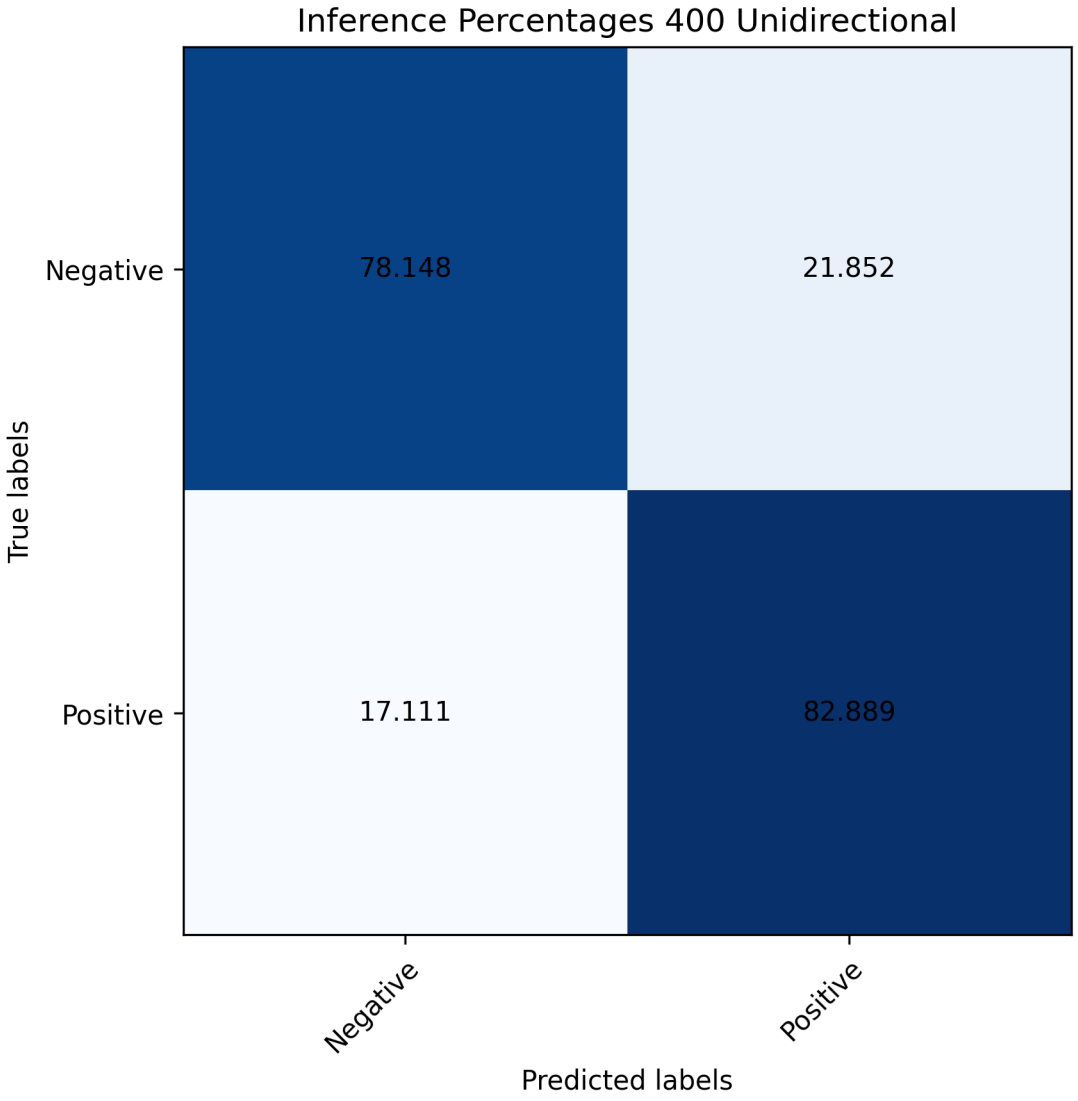
2.0.7 Figure 13. Testing Results Confusion Matrix for Unidirectional 200 Embeddings



2.0.8 Figure 14. Testing Results Confusion Matrix for Bidirectional 200 Embeddings



2.0.9 Figure 15. Testing Results Confusion Matrix for Unidirectional 400 Embeddings



2.0.10 Figure 16. Testing Results Confusion Matrix for Bidirectional 400 Embeddings